# Development Workflow mit Maven

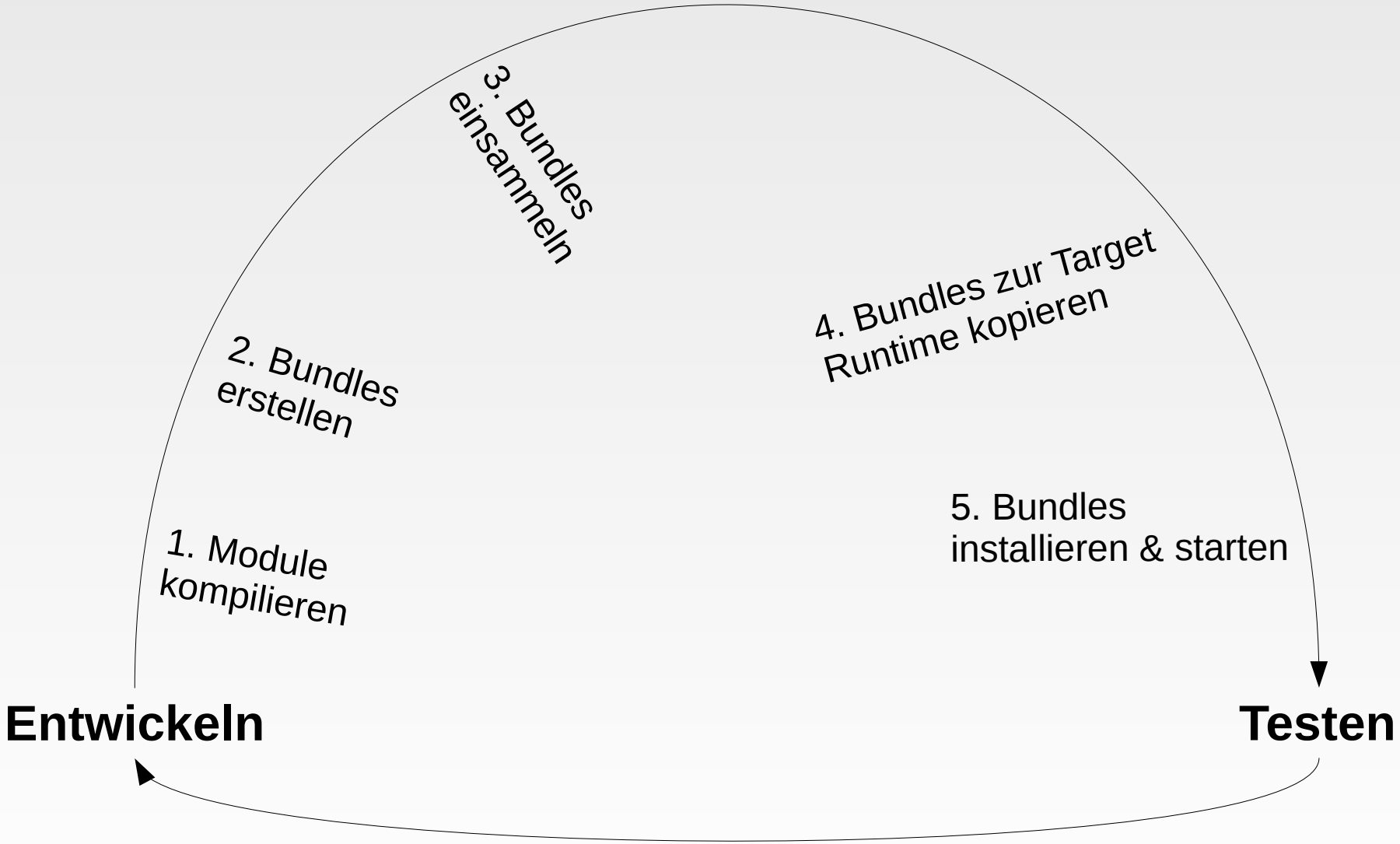
Roman Roelofsen
Lead Architect

@

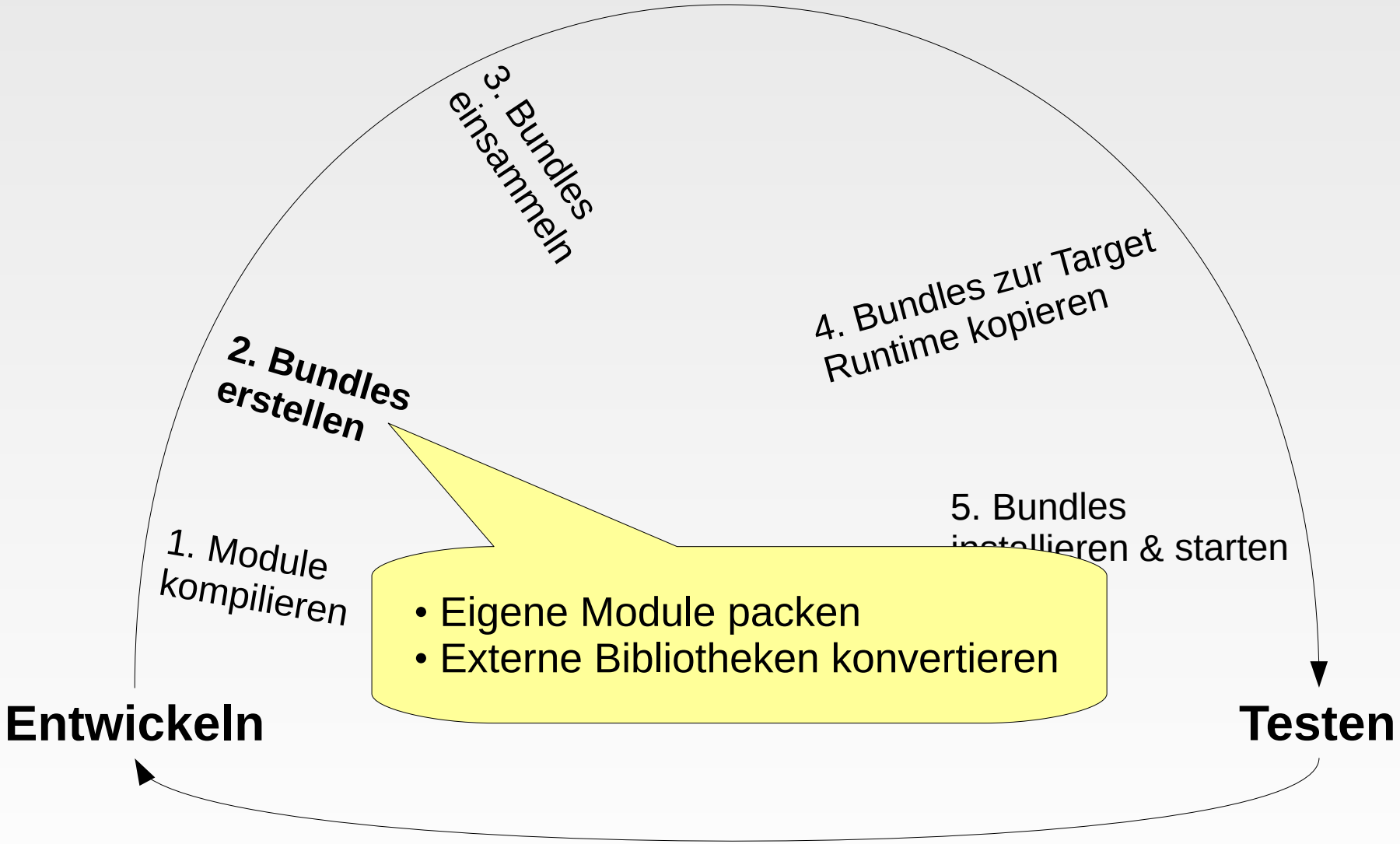
# Ein typisches OSGi Projekt

- # eigene Bundles
- # externe Bibliotheken (Bundle vorhanden)
- # externe Bibliotheken (Bundle nicht vorhanden)
- # Target Runtimes

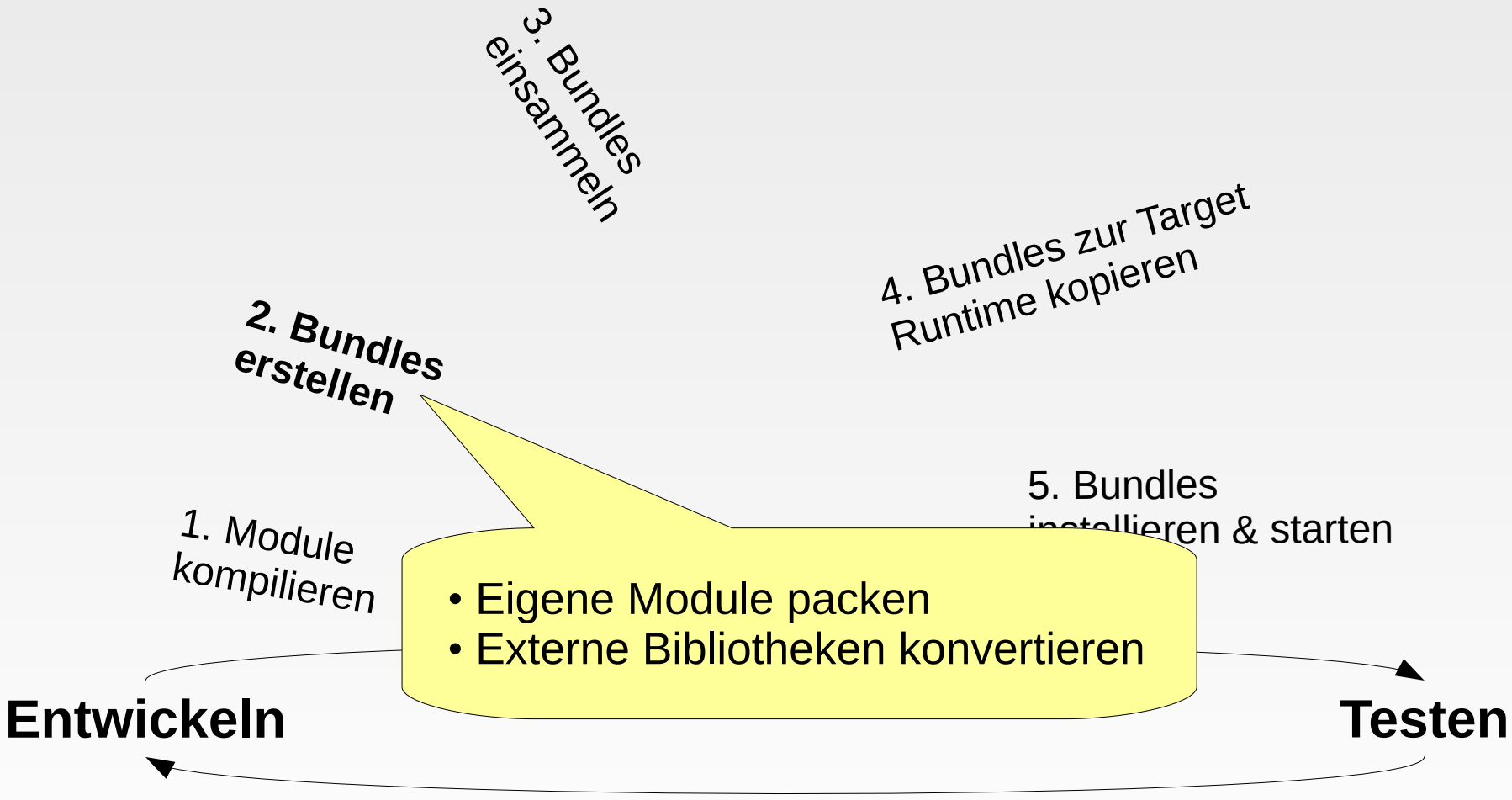
# Development Workflow



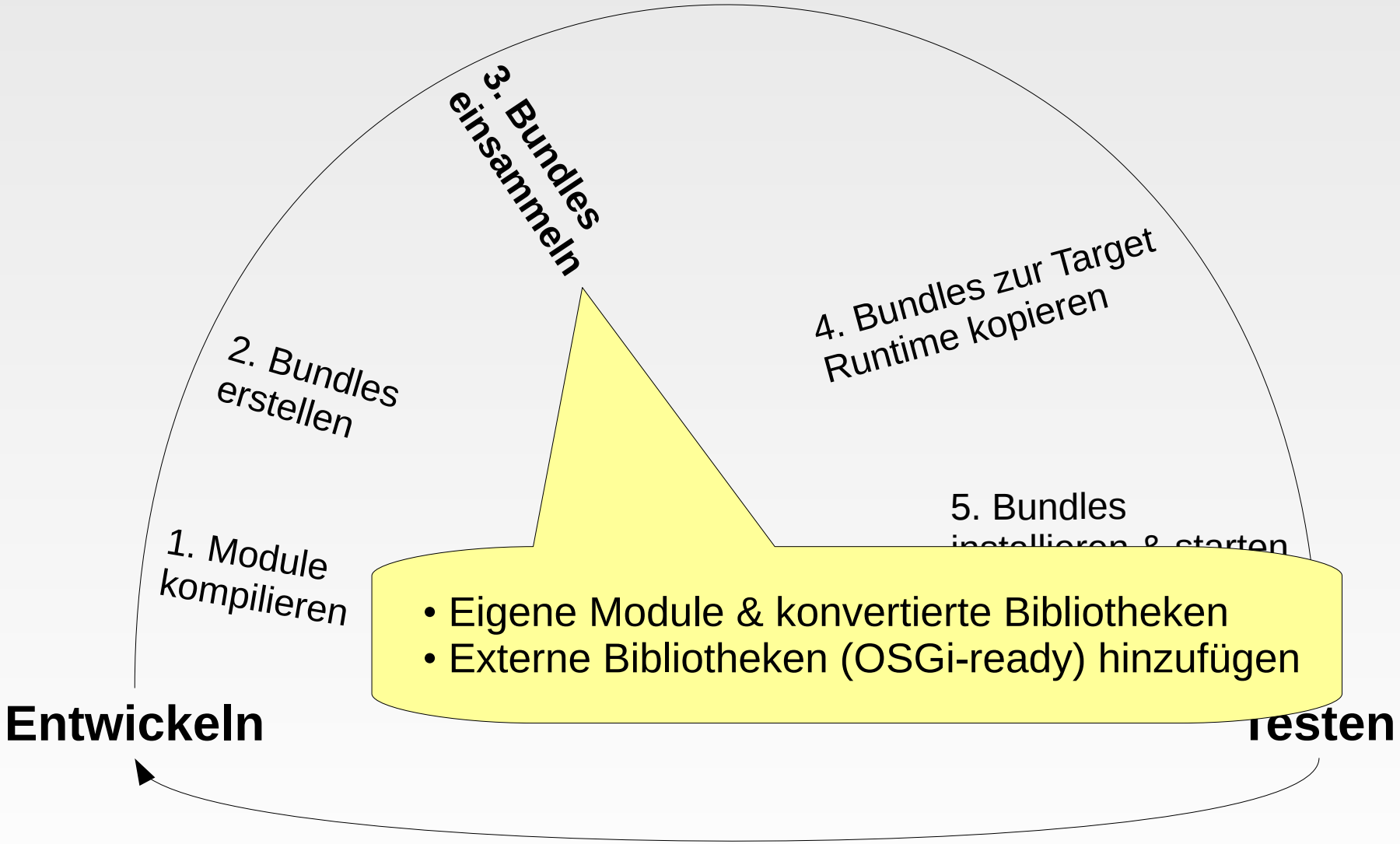
# Development Workflow



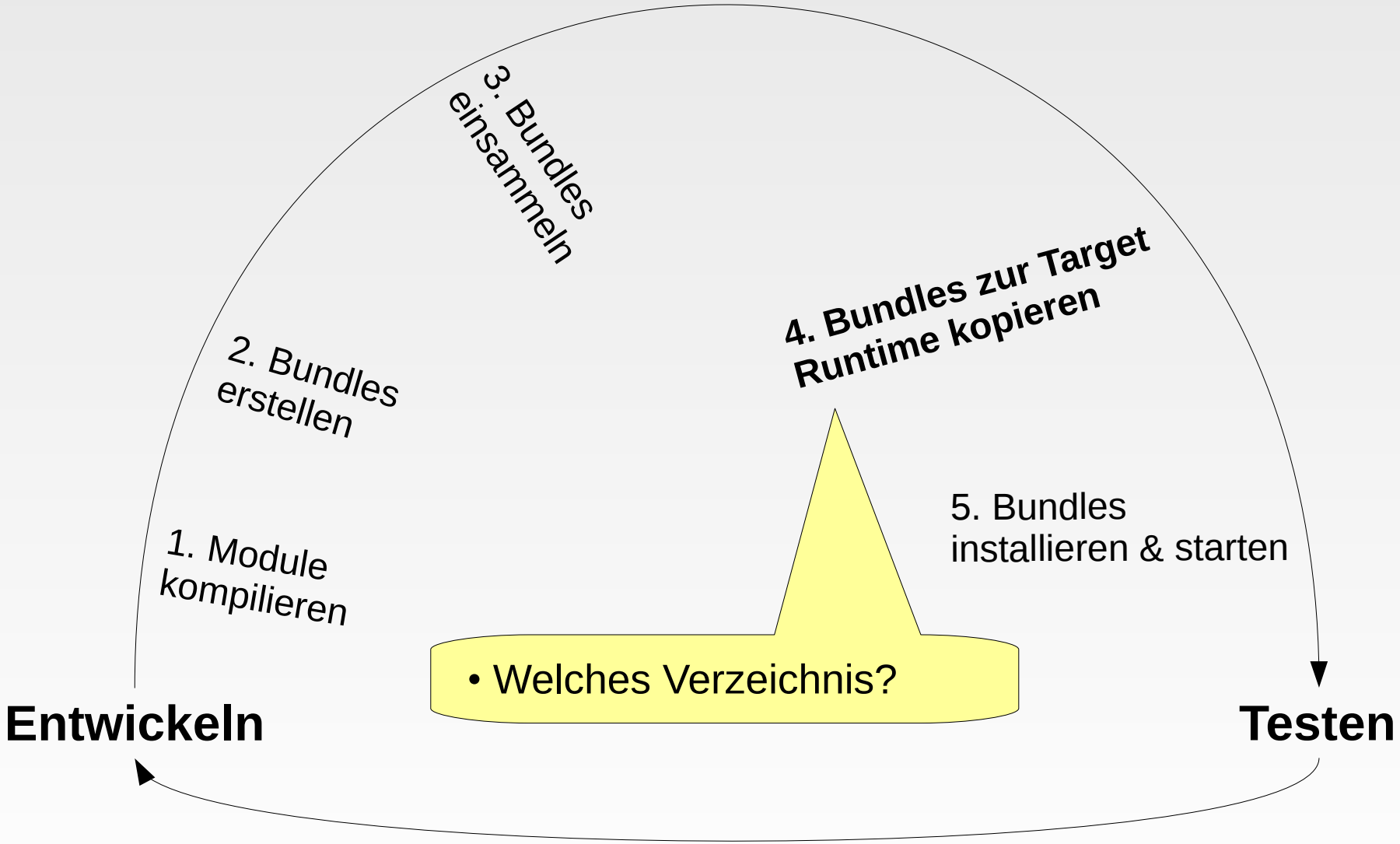
# Development Workflow



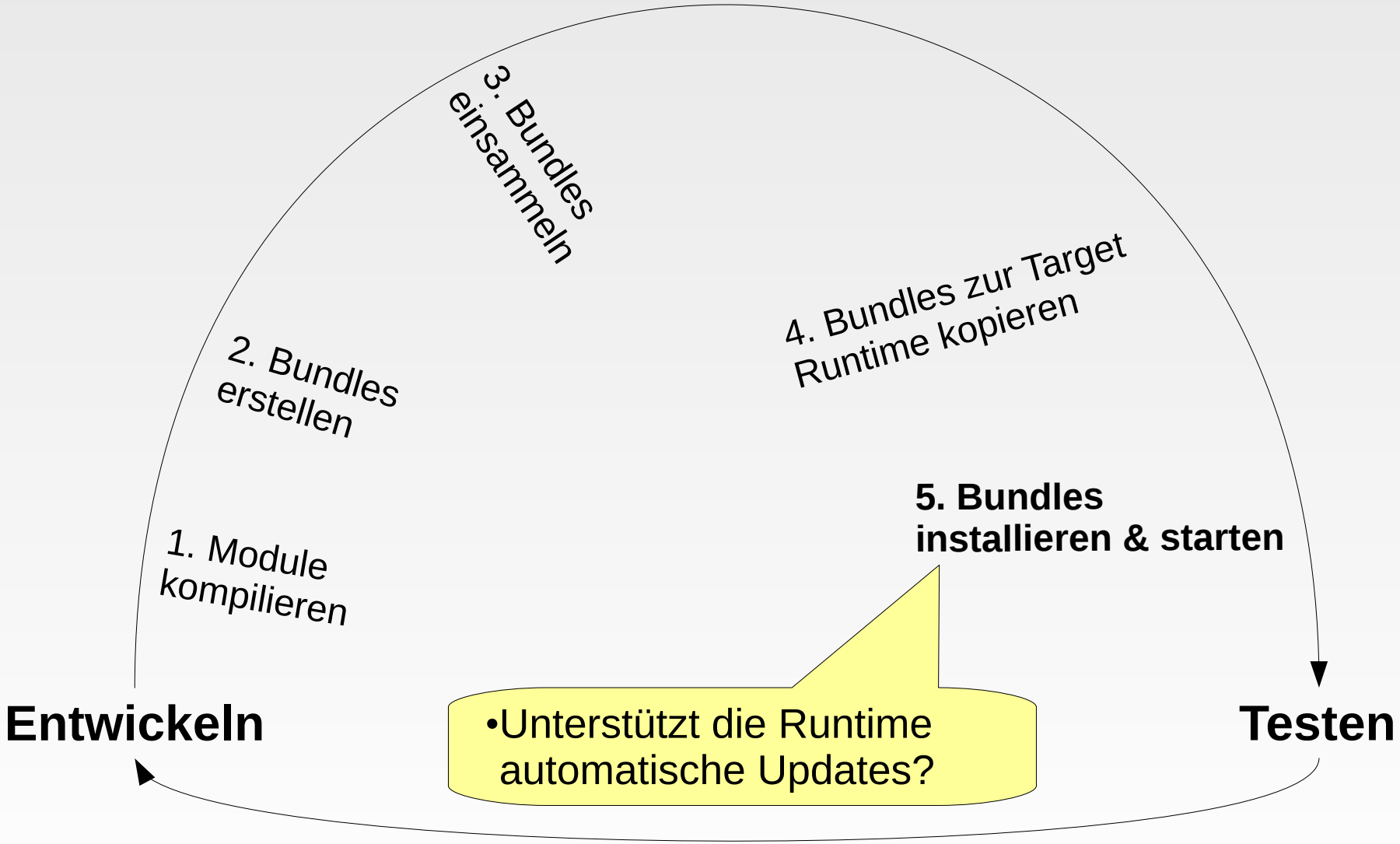
# Development Workflow



# Development Workflow



# Development Workflow



# Zusätzliche Anforderungen

- Build muss unverändert Standalone funktionieren
- Kein manuelles Kopieren von JAR Dateien
- Alle Abhängigkeiten müssen durch Repositories bedient werden
- Keine Redundanz bei z.B. Versionsangaben
- Je mehr OSGi-Unterstützung, desto besser
  - Export mit uses-Direktive
  - Import mit Versionsbereichen
  - Warnung bei split packages

# Bundles erstellen: Eigene Module

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <configuration>
        <instructions>
            <Private-Package>
                acme.bundle1.*
            </Private-Package>
            <Export-Package>
                acme.bundle1.api.*;version=${project.version}
            </Export-Package>
            ...
```

# Bundles erstellen: Externe Bibliotheken konvertieren

```
<dependency>
    <groupId>org.mortbay</groupId>
    <artifactId>jetty</artifactId>
    <version>${jetty.version}</version>
</dependency>

<Export-Package>
    org.mortbay.jetty.*;version=${jetty.version}
</Export-Package>
```

# Bundles erstellen: Externe Bibliotheken einbetten

```
<dependency>
    <groupId>org.mortbay</groupId>
    <artifactId>jetty</artifactId>
</dependency>

<Private-Package>
    acme.bundle1.*,
    org.mortbay.jetty.*
</Private-Package>

<Export-Package>
    acme.bundle1.api.*;version=${project.version}
</Export-Package>
```

# Bundles erstellen: Vorteile BND

- Rekursive Exports
- Automatische Erstellung von Imports für Exports
- Uses-Direktive wird automatisch erstellt
- Imports werden automatisch berechnet
  - Basierend auf Bytecode
  - Support für u.a. Spring Konfigurationen
- Versionsbereiche können zum Teil mit versionpolicy automatisch erstellt werden
  - `[${version;==;${@}},${version;=+;${@}})`
- Einheitliche Sicht auf classpath
  - Granularität der Bundles kann leicht verändert werden
- Erweiterbar (Plugins)

# Bundles testen

- Integrationstests sollten das Zusammenspiel der Bundles testen
  - Resolving
  - Service-Abhängigkeiten
  - Extender-Pattern
- Wichtige Anforderungen
  - Keine Redundanz der Versionsinformationen
  - Keine eigene Test-API

# Bundles testen

- Pax Exam Abhängigkeiten hinzufügen

```
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam</artifactId>
</dependency>
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-junit</artifactId>
</dependency>
<dependency>
  <groupId>org.ops4j.pax.exam</groupId>
  <artifactId>pax-exam-container-default</artifactId>
</dependency>
```

# Bundles testen

- Pax Exam Plugin einbinden

```
<plugin>
   <groupId>org.ops4j.pax.exam</groupId>
   <artifactId>maven-paxexam-plugin</artifactId>
   <configuration>
     <options>
       <autoWrap>true</autoWrap>
       <platform>equinox</platform>
     </options>
   </configuration>
 </plugin>
```

# Bundles testen

- Pax Exam Plugin einbinden

```
@RunWith( MavenConfiguredJUnit4TestRunner.class )
public class NewPaxExamFeature {
    @Inject BundleContext context;

    @Test
    public void blaBla() {
        ...
    }
}
```

# Bundles einsammeln

- Maven Assembly Plugin
- Module & Abhängigkeiten werden über Maven Koordinaten identifiziert
- JARs und normale Dateien können beliebig kopiert werden
- Ergebnis kann gezippt werden etc.

# Bundles einsammeln

```
<dependencySet>
    <includes>
        <include>org.apache.felix:org.apache.felix.main</include>
    </includes>
    <outputDirectory>bin</outputDirectory>
</dependencySet>

<dependencySet>
    <excludes>
        <exclude>org.apache.felix:org.apache.felix.main</exclude>
    </excludes>
    <outputDirectory>bundle</outputDirectory>
</dependencySet>
```

# Target Runtime

- Target Runtime ist oft schon auf dem Entwickler-PC vorhanden
- Eventuell ist jedoch eine Installation im Rahmen des Builds wünschenswert
  - Reproduzierbar
  - Distribution
- Kann mit diversen Plugins (z.B. antrun) automatisiert werden

# Target Runtime

```
<artifactId>maven-antrun-plugin</artifactId>
<executions><execution>
<id>generate-resources</id>
<phase>generate-resources</phase>
<configuration>
  <tasks>
    <mkdir dir="target/downloaded" />
    <get src="${karaf.url}" dest="target/downloaded/karaf.tar.gz" ... />
    <gunzip src="target/downloaded/karaf.tar.gz" />
    <untar src="target/downloaded/karaf.tar" dest="target/generated/runtime" .. />
  </tasks>
</configuration>
<goals><goal>run</goal></goals>
</execution></executions>
```

# Bundles zur Target Runtime kopieren

- Hängt sehr von der Umgebung des Entwicklers ab
- Alle Bundles müssen in das Deployment-Verzeichnis der Target Runtime kopiert werden
- Kann mit z.B. maven-antrun-plugin automatisiert werden
- Eventuell ist ein manueller Eingriff nötig
  - Redundante Bundles
  - Unterschiedliche Verzeichnis für verschiedene Bundles

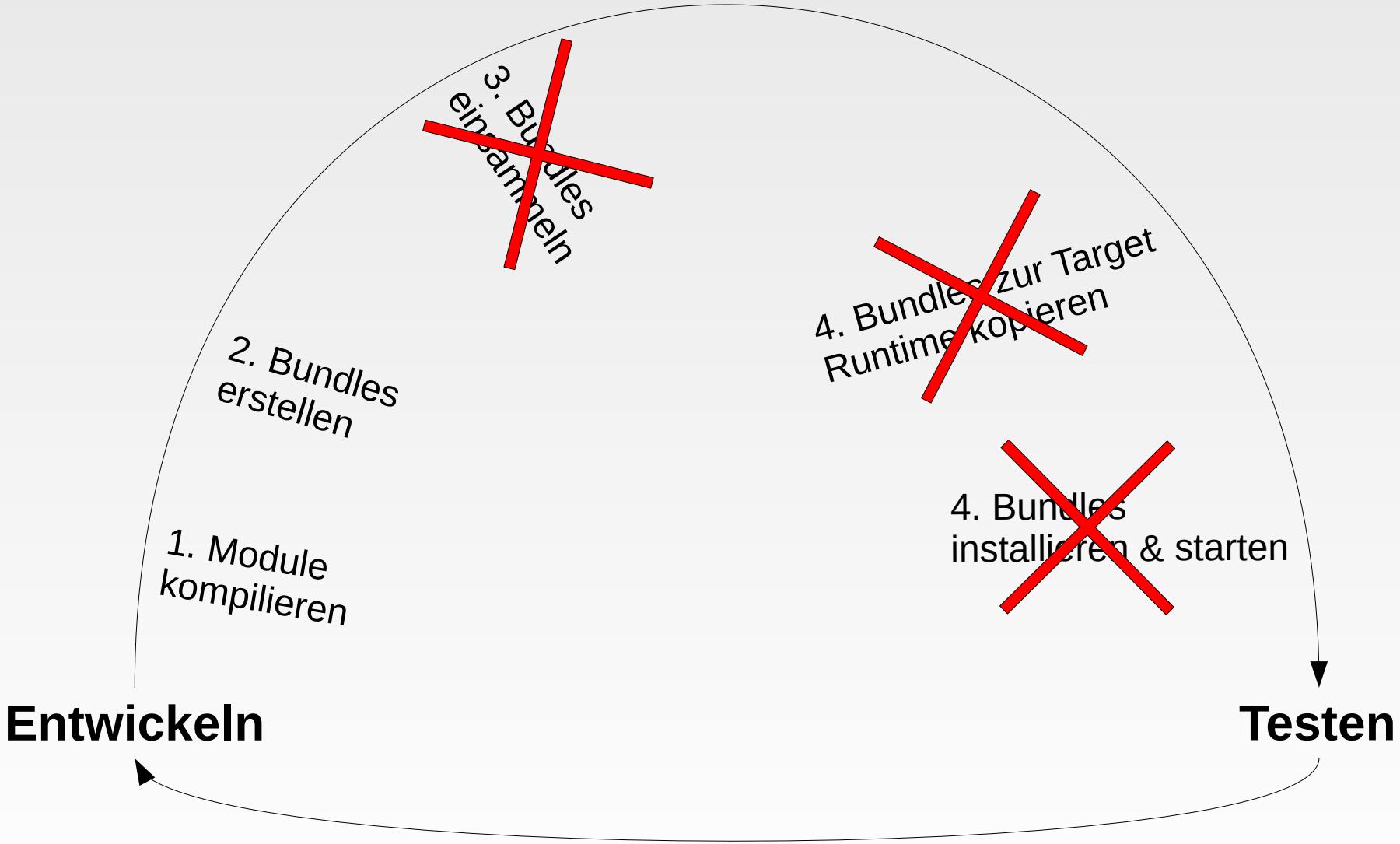
# Target Runtime kopieren

```
<fileSets>
  <fileSet>
    <directory>src/main/resources/runtime</directory>
    <outputDirectory></outputDirectory>
  </fileSet>
  <fileSet>
    <directory>target/generated/runtime</directory>
    <outputDirectory></outputDirectory>
  </fileSet>
</fileSets>
```

# Bundles installieren & starten

- Hängt sehr von der Umgebung des Entwicklers ab
- Target Runtime muss Hot-Deployment unterstützen
- Die nötige API ist jedenfalls in OSGi vorhanden

# The Bundle Buddy (TBB)



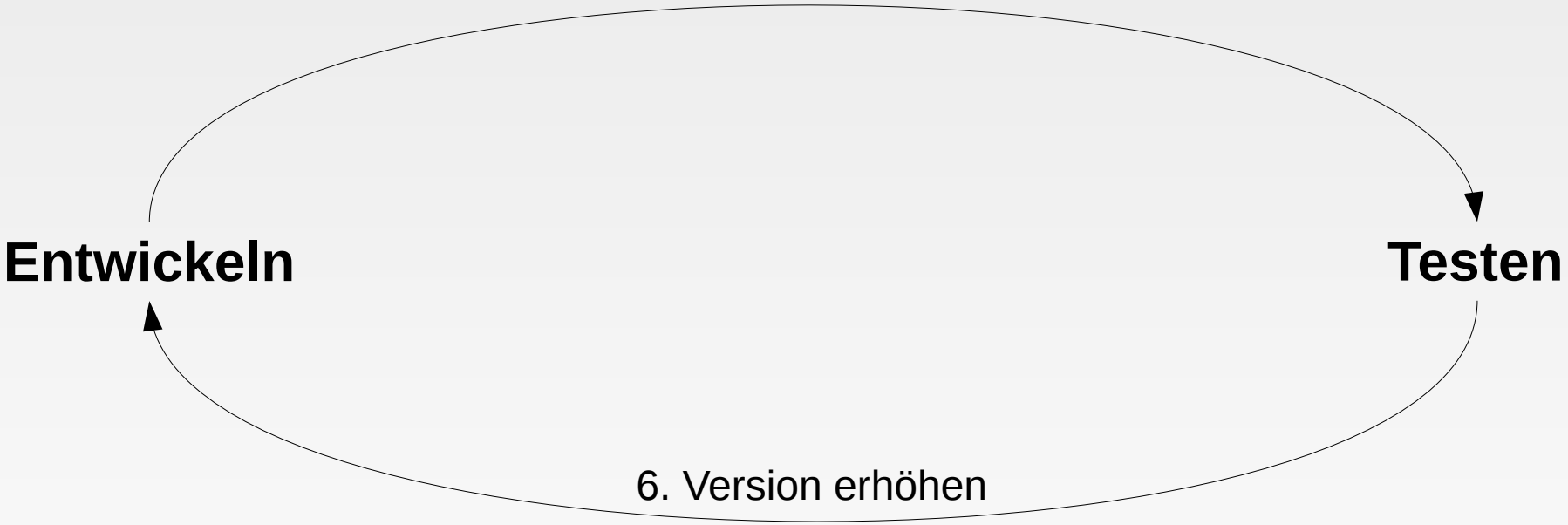
# The Bundle Buddy (TBB)

- System- oder Framework-Property „tbb“ spezifiziert das Projekt-Verzeichnis
  - `export tbb=/home/roman/workspace/kasse`
- TBB Bundle wird in der Target Runtime installiert
- Im Projekt-Verzeichnis werden alle Bundles identifiziert und mit den in der Target Runtime installierten abgeglichen
  - Bundle-SymbolicName + Bundle-Version
- Ändert sich eine Bundle JAR Datei im Projekt-Verzeichnis wird das installierte Bundle automatisch aktualisiert

# The Bundle Buddy (TBB)

DEMO

# Development Workflow



# Version erhöhen

- Nachdem eine Änderung erfolgreich getestet wurde, sollte die Version des Bundles erhöht werden
- Abhängig vom im Team definierten Prozess
- Versionserhöhung kann an ein Maven Goal (z.B. install oder deploy) gebunden werden
- Kann mit gmaven-plugin + versions-maven-plugin automatisiert werden

# Version erhöhen

- Mit Groovy eine neue Version erzeugen

```
<groupId>org.codehaus.groovy.maven</groupId>
<artifactId>gmaven-plugin</artifactId>
<executions><execution>
  <phase>verify</phase><goals><goal>execute</goal></goals>
  <configuration>
    <source>
      lastDot = project.version.lastIndexOf(".");
      Left = project.version.substring(0, lastDot + 1);
      Right = project.version.substring(lastDot + 1).toInteger() + 1;
      project.properties.genNewVersion = left + right;
    </source>
  </configuration>
</execution></executions>
```

# Version erhöhen

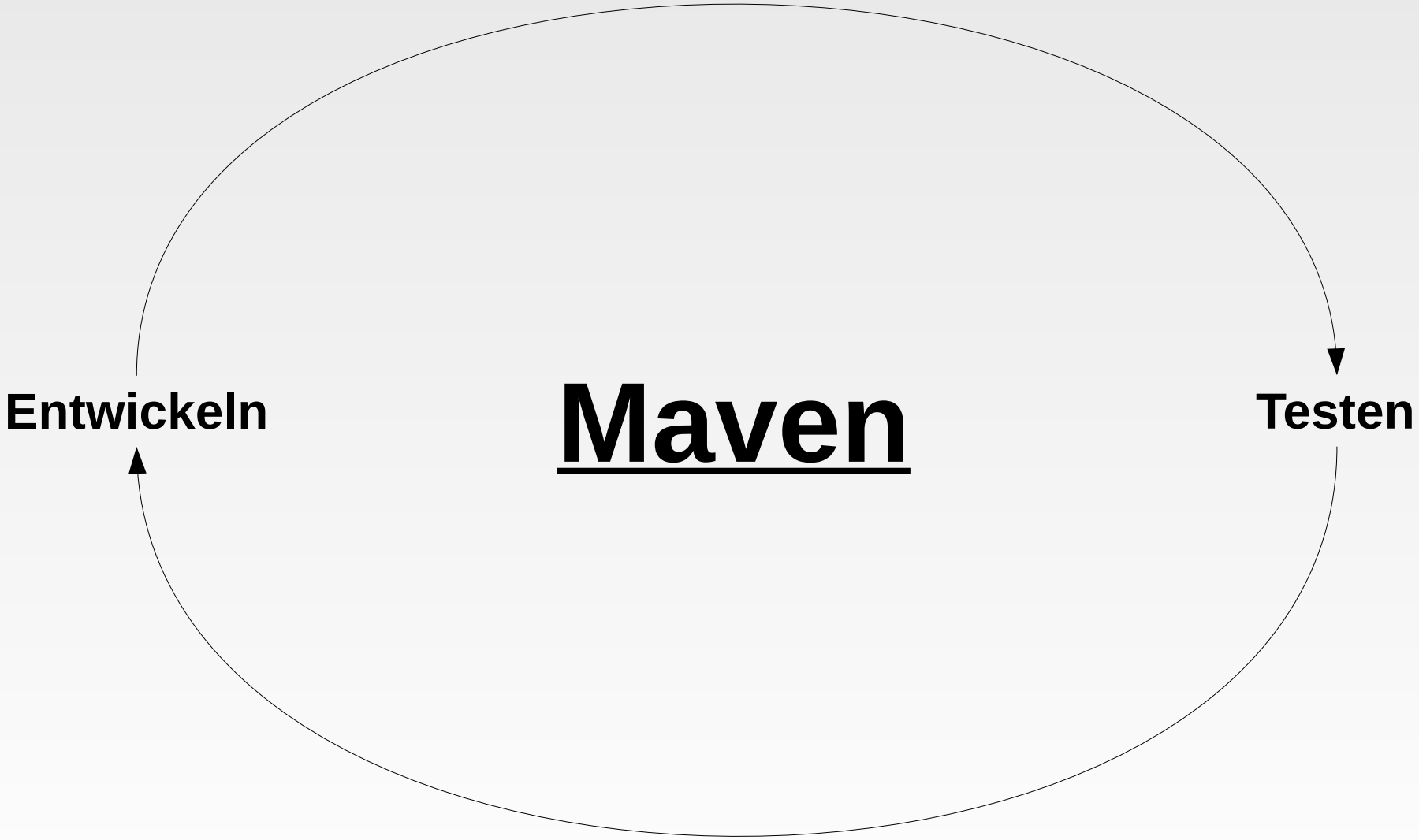
- Mit versions-maven-plugin die Version automatisch setzen

```
<groupId>org.codehaus.mojo</groupId>
<artifactId>exec-maven-plugin</artifactId>
<executions><execution>
  <phase>install</phase>
  <goals><goal>exec</goal></goals>
</execution></executions>
<configuration>
  <executable>mvn</executable>
  <arguments>
    <argument>versions:set</argument>
    <argument>-DnewVersion=${genNewVersion}</argument>
    </arguments>
</configuration>
```

# Version erhöhen

DEMO

# Fazit



# Vielen Dank für Ihre Aufmerksamkeit!

Roman Roelofsen
r.roelofsen@prosyst.com
Twitter: romanroe